# ADIFOR: A FORTRAN SYSTEM FOR PORTABLE AUTOMATIC DIFFERENTIATION*

**Christian Bischof**
**Andreas Griewank**

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, Illinois 60439

Preprint MCS-P317-0792

## Abstract

Automatic differentiation provides the foundation for sensitivity analysis and subsequent design optimization of complex systems by reliably computing derivatives of large computer codes, with the potential of doing it many times faster compared to current approaches. This paper describes the ADIFOR (Automatic DIfferentiation of FORtran) system, a translator that augments Fortran programs with statements for the computation of derivatives. ADIFOR accepts arbitrary Fortran 77 code defining the computation of a function and writes portable Fortran 77 code for the computation of its derivatives. Our goal is to free the computational scientist from worrying about the accurate and efficient computation of derivatives, even for complicated "functions", thereby enabling him to concentrate on the more important issues of system modeling and algorithm design. This paper gives an overview of the principles underlying the ADIFOR system, and comments on the power of automatic differentiation for computing derivatives of implicitly-defined functions.

## 1 Introduction

The modeling of complex systems typically involves an analysis or simulation phase where the stationary state or the time evolution of the computer model is determined from given parameters and initial conditions. Subsequently, the computational results are compared with observed data or analyzed in view of theoretical insights and expectations. These considerations are likely to suggest numerical parameter adjustments or even structural modifications in the computational model.

In this process, it is critical to assess how changes in the input parameters affect changes in the output. Derivatives quantify this effect, and the task of analyzing and adjusting the computational model usually relies heavily on the availability of so-called *sensitivities*, namely, the partial derivatives of intermediate or final results with respect to input parameters or other intermediate variables. In any case, the two-stage process of sensitivity analysis and subsequent model adjustment is likely to be repeated several times before the computational results are judged to reflect the modeled system state or yield a useful prediction of its time evolution sufficiently closely. It should be obvious that the accuracy of the computed sensitivities is critical in guiding this adjustment and validation process.

However, in most applications, the sensitivity analysis of such an analysis code is only the first step. Once the model has been validated, one is interested in determining the values of the input parameters of the computational model such as to achieve a desired physical behavior. For example, a major thrust of the Computational Aerosciences project of NASA's High Performance Computing and Communications Program is multidisciplinary design and optimization of a high speed civil transport [19]. A related activity is the HiSAIR (High Speed Airframe Integration Research) project which also has the optimization of a high speed civil transport as a focus and encompasses a greater breadth of technical disciplines influencing the design of such a vehicle.

A key technology that is required for these optimization procedures is the capability to calculate the sensitivity derivatives of outputs from the various analysis codes with respect to a set of design variables that are adjusted by a nonlinear programming code to achieve some optimal measure of vehicle performance. Other applications arise in real-time simulation and control

*This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U. S. Department of Energy, under Contract W-31-109-Eng-38 and by the National Science Foundation under Cooperative Agreement Number CCR-9120008.

This paper appeared in the Proceedings of the 4th Symposium on Multidisciplinary Analysis and Optimization, AIAA Paper 92-4744, pages 433–441, 1992.

of road vehicles, robots, and macromolecules, parameter identification problems in geophysical systems and accelerator beam tracing for the design of optical instruments and particle accelerators [17].

There are four approaches to computing derivatives:

**By Hand:** This is error-prone, and applicable only in simple cases.

**Divided Differences:** The derivative of $f$ with respect to the $i$th component of $x$ at a particular point $x_o$ is approximated by either *one-sided differences*

$$\frac{\partial f(x)}{\partial x_i}|_{x=x_o} \approx \frac{f(x_o \pm h * e_i) - f(x_o)}{\pm h}$$

or *central differences*

$$\frac{\partial f(x)}{\partial x_i}|_{x=x_o} \approx \frac{f(x_o + h * e_i) - f(x_o - h * e_i)}{2h}.$$

Here $e_i$ is the $i$th cartesian basis vector. Computing derivatives by divided differences has the advantage that the function is only needed as a 'black box'. The main drawback of divided differences is that their accuracy is hard to assess. A small step size $h$ is needed for properly approximating derivatives, yet may lead to numerical cancellation and the loss of many digits of accuracy. In addition, varying scales of the $x_i's$ may require different step sizes for the various parameters.

**Symbolic Differentiation:** This functionality is provided by symbolic manipulation packages such as Maple, Reduce, Macsyma, or Mathematica. Given a string describing the definition of a function, symbolic manipulation packages provide exact derivatives, expressing the derivatives all in terms of the intermediate variables. Symbolic differentiation is a powerful technique, but quickly runs into resource limitations when applied to even moderately sized problems (described by 100 lines of code, say).

**Automatic Differentiation:** Automatic differentiation techniques rely on the fact that every function, no matter how complicated, is executed on a computer as a (potentially very long) sequence of elementary operations such as additions, multiplications, and elementary functions such as sin and cos. By applying the chain rule, e.g.

$$\frac{\partial}{\partial t} f(g(t))|_{t=t_0} = \left(\frac{\partial}{\partial s} f(s)|_{s=g(t_0)}\right)\left(\frac{\partial}{\partial t} g(t)|_{t=t_0}\right)$$

over and over again to the composition of those elementary operations, one can compute derivative information of $f$ exactly and in a completely mechanical fashion.

```
if x(1) > 2 then
   a = x(1)+x(2)
else
    a = - x(1)/(x(2)*x(2)*x(2))
end if
do i = 1, 2
    a = a*x(i)
end do
y(1) = a/x(2)
y(2) = sin(x(2))
```

Figure 1: Sample Program for a Function $f : x \mapsto y$

The paper is structured as follows. The next section gives an introduction into automatic differentiation, and in section 3 we give a brief overview of the functionality of the ADIFOR automatic differentiation system. Section 4 elaborates on the connection between automatic differentiation and the computation of derivatives of implicitly defined functions. Lastly, we comment on some future improvements of ADIFOR.

## 2 Automatic Differentiation

We illustrate automatic differentiation with an example. Assume that we have the sample program shown in Figure 1 for the computation of a function $f : \mathbf{R}^2 \to \mathbf{R}^2$. Here, the vector $\mathtt{x}$ contains the independent variables, and the vector $\mathtt{y}$ contains the dependent variables. The function described by this program is defined except at $\mathtt{x(2)} = 0$ and is differentiable except at $\mathtt{x(1)} = 2$.

By associating a derivative object $\nabla\mathtt{t}$ with every variable $\mathtt{t}$, we can transform this program into one for computing derivatives. Assume that $\nabla\mathtt{t}$ contains the derivatives of $\mathtt{t}$ with respect to the independent variables $\mathtt{x}$,

$$\nabla\mathtt{t} = \begin{pmatrix} \frac{\partial\,\mathtt{t}}{\partial\,\mathtt{x(1)}} \\ \frac{\partial\,\mathtt{t}}{\partial\,\mathtt{x(2)}} \end{pmatrix}.$$

We can propagate those derivatives by using elementary differentiation arithmetic based on the chain rule (see for example [23] for more details). For example, the statement

$$\mathtt{a} = \mathtt{x(1)} + \mathtt{x(2)}$$

implies

$$\nabla\mathtt{a} = \nabla\mathtt{x(1)} + \nabla\mathtt{x(2)}.$$

The chain rule, applied to the statement

$$\mathtt{y(1)} = \mathtt{a}/\mathtt{x(2)},$$

implies that

$$\begin{aligned} \nabla\mathtt{y(1)} &= \frac{\partial\,\mathtt{y(1)}}{\partial\,\mathtt{a}} * \nabla\mathtt{a} + \frac{\partial\,\mathtt{y(1)}}{\partial\,\mathtt{x(2)}} * \nabla\mathtt{x(2)} \\ &= \mathtt{1.0}/\mathtt{x(2)} * \nabla\mathtt{a} - (\mathtt{a}/(\mathtt{x(2)} * \mathtt{x(2)})) * \nabla\mathtt{x(2)}. \end{aligned}$$

```
t1 = - y
t2 = z * z
t3 = t2 * z
w = t1 / t3
t1bar = (1 / t3)
t3bar = (- t1 / t3)
t2bar = t3bar * z
zbar  = t3bar * t2
zbar  = zbar + t2bar * z
zbar  = zbar + t2bar * z
ybar  = - t1bar
```

Figure 2: Reverse Mode Computation of $(\mathtt{ybar}, \mathtt{zbar}) = (\frac{\partial w}{\partial y}, \frac{\partial w}{\partial z})$

Elementary functions are easy to deal with. For example, the statement

$$\mathtt{y(2)} = \mathtt{sin(x(2))}$$

implies

$$\nabla \mathtt{y(2)} = \frac{\partial\, \mathtt{y(2)}}{\partial\, \mathtt{x(2)}} * \nabla \mathtt{x(2)} = \mathtt{cos(x(2))} * \nabla \mathtt{x(2)}.$$

Straightforward application of the chain rule in this fashion then leads to the so-called *forward mode* of automatic differentiation, which propagates *the derivatives with respect to the independent variables.*

Another way of automatic differentiation is the so-called *reverse mode*, which propagates *the derivatives of the final result with respect to an intermediate quantities.* These quantities are usually called *adjoints.* For illustration sake, let us consider the statement

$$\mathtt{w} = -\mathtt{y}/(\mathtt{z} * \mathtt{z} * \mathtt{z}).$$

In the reverse mode, let `tbar` denote the adjoint object corresponding to `t`. The goal is for `tbar` to contain the derivative $\frac{\partial w}{\partial t}$. We know that $\mathtt{wbar} = \frac{\partial w}{\partial w} = 1.0$. We can compute `ybar` and `zbar` by applying the following simple rule to the binary statements executed in computing `w`, but in reverse order:

```
if s = f(t), then tbar += sbar * (df / dt)
if s = f(t,u), then tbar += sbar * (df /dt)
                    ubar += sbar * (df /du)
```

Using this simple recipe (see [15, 23]), and applying straightforward optimizations (see [2] for a more detailed description), we generate the so-called adjoint code shown in Figure 2 for computing `w` and its derivatives.

If the statement for the computation of `w` was embedded into a larger computation, one can then exploit the fact that

$$\nabla \mathtt{w} = \frac{\partial\, \mathtt{w}}{\partial\, \mathtt{y}} * \nabla \mathtt{y} + \frac{\partial\, \mathtt{w}}{\partial\, \mathtt{z}} * \nabla \mathtt{z}.$$

```
if x(1) > 2.0 then
   ∇a = ∇x(1) + ∇x(2);
   a = x(1)+x(2);
else
   t1 = - x(1); t2 = x(2) * x(2);
   t3 = t2 * x(2); t1bar = (1 / t3);
   t3bar = (- t1 / t3); t2bar = t3bar * z;
   zbar = t3bar * t2; zbar = zbar + t2bar * z;
   zbar = zbar + t2bar * z; ybar = - t1bar;
   ∇a = ybar * ∇x(1) + zbar * ∇x(2)
   a = t1/t3;
end if
do i = 1, 2
   ∇a = x(i) * ∇a + a * ∇x(i);
   a = a * x(i);
end do
∇y(1) = 1.0/x(2) * ∇a - a/(x(2)*x(2)) * ∇x(2);
y(1) = a/x(2);
∇y(2) = cos(x(2)) * ∇x(2);
y(2) = sin(x(2))
```

Figure 3: Sample Program of Figure 1 Augmented with Derivative Statements

Hence, after computing the "local" derivatives $(\frac{\partial w}{\partial y}, \frac{\partial w}{\partial z})$ of `w` with respect to `z` and `y`, one can then easily compute $\nabla \mathtt{w}$, the derivatives of `w` with respect to `x`. That is, we "preaccumulate" the derivatives of the right-hand side, and then use the chain rule to advance the global computation. This hybrid approach saves storage and computation compared to the straightforward forward mode whenever the number of variables on the right-hand side of an assignment statement is greater than two. Using this hybrid approach on the code shown in Figure 1 we arrive at the pseudo-code shown in Figure 3 for computing the derivatives of `y(1)` and `y(2)`.

The derivatives computed by automatic differentiation are guaranteed to be reliable, unlike those computed by divided difference approximations. Griewank and Reese [18] have shown that in the presence of round-off the derivative objects computed by automatic differentiation are the exact result of a nonlinear system whose elementary partial derivatives have been perturbed by factors of at most $(1+\varepsilon)^2$, where $\varepsilon$ is the relative machine precision.

We also mention that the automatic differentiation approach can easily be generalized to the computation of univariate Taylor series or multivariate higher-order derivatives [10,23,16,4].

## 3 The ADIFOR Automatic Differentiation Tool

The automatic differentiation of computer arithmetic has been investigated since before 1960. Since then there have been various implementations of auto-

matic differentiation, and a recent survey can be found in [20]. However, for the most part, they were conceived by the need for accurate first- and higher-order derivatives in a certain application. Distribution for the mainstream of scientific computing was not a major concern, and, since these tools for the most part computed derivatives slower than divided difference approaches, potential users were discouraged.

Recently, however, process towards a general-purpose automatic differentiation tool competitive with divided differences has been made with the development of ADIFOR (Automatic Differentiation in Fortran) [2, 5, 3, 1]. ADIFOR provides automatic differentiation for programs written in Fortran 77. Given a Fortran subroutine (or collection of subroutines) describing a "function", and an indication which variables in parameter lists or common blocks correspond to "independent" and "dependent" variables with respect to differentiation, ADIFOR produces Fortran 77 code that allows the computation of the derivatives of the dependent variables with respect to the independent ones.

ADIFOR was designed from the outset with large-scale codes in mind, and it uses the facilities of the ParaScope Fortran environment [7,8] to parse the code, and extract control flow and dependence flow information. ADIFOR produces portable Fortran-77 code and accepts almost all of Fortran-77, in particular arbitrary calling sequences, nested subroutines, common blocks and equivalences. The ADIFOR-generated code tries to preserve vectorization and parallelism in the original code, and employs a consistent subroutine naming scheme which allows for code tuning, the exploitation of domain-specific knowledge and the exploitation of vendor-supplied libraries.

ADIFOR employs the hybrid forward/reverse mode that was shown in the previous section. That is, for each assignment statement, we generate code for computing the partial derivatives of the result with respect to the variables on the right-hand side, and then employ the forward mode to propagate overall derivatives. The resulting decrease in complexity compared to a straightforward forward mode implementation usually is substantial. For example, the code for the blunt body shock tracking problem by Shubin [25] needs to compute the $190 \times 190$ Jacobian of a "function" described by 1400 lines of Fortran code. When we execute this code for a particular set of input values, we execute a total of 1840 assignment statements that were augmented with derivative computations. The distribution of the number of variables in the right-hand side of the assignment statements is shown in Figure 4. We see that only 543 assignments involve two variables, and as a result the hybrid mode used in ADIFOR computes derivatives at roughly 69% the cost of a straightforward forward mode implementation.

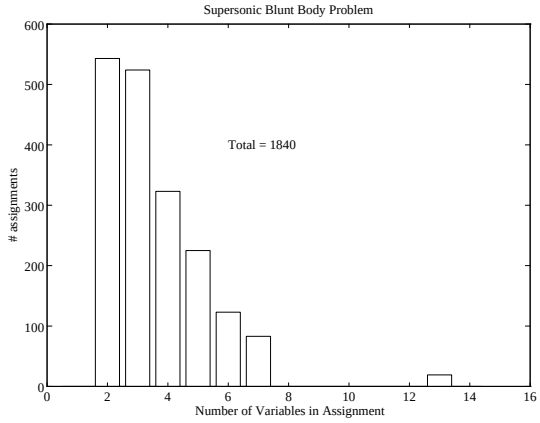


Figure 4: Number of Variables in Assignment Statements

We also stress that ADIFOR uses the data flow analysis information from ParaScope to determine the set of variables that require derivative information in addition to the dependent and independent ones. This approach allows for an intuitive interface, and greatly reduces the storage requirements of the derivative code.

ADIFOR-generated code can be used in various ways: Instead of simply producing code to compute the Jacobian $J$, ADIFOR produces code to compute $J * S$, where the "seed matrix" $S$ is initialized by the user. So if $S$ is the identity, ADIFOR computes the full Jacobian, and if $S$ is just a vector, ADIFOR computes the product of the Jacobian by a vector. "Compressed" versions of sparse Jacobians can be computed by exploiting the same graph coloring techniques [12, 11] that are used for divided difference approximations of sparse Jacobians.

The idea is best understood with an example. Assume that we have a function

$$F = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \end{pmatrix} : x \in \mathbf{R}^4 \mapsto y \in \mathbf{R}^5$$

whose Jacobian $J$ has the following structure (symbols denote nonzeros, and zeros are not shown):

$$J = \begin{pmatrix} \bigcirc & & & \\ \bigcirc & & & \diamond \\ & \triangle & & \diamond \\ & \triangle & \square & \\ & \triangle & \square & \end{pmatrix}.$$

So columns 1 and 2, as well as columns 3 and 4 are structurally orthogonal, and in divided-difference approximations one could exploit that by perturbing both


Figure 5: Sparsity Structure of $190 \times 190$ Blunt Body Problem Jacobian


Figure 6: Sparsity Structure of Compressed $190 \times 28$ Blunt Body Problem Jacobian

$x_1$ and $x_2$ in one function evaluation, and both $x_3$ and $x_4$ in the other. In ADIFOR we exploit this fact by setting

$$S = \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{pmatrix},$$

For a more realistic example, the $190 \times 190$ Jacobian of Shubin's blunt body shock tracking problem has only 2582 nonzero entries and its structure is shown in Figure 5. Due to its sparsity structure, it can be condensed into the "compressed Jacobian" shown in Figure 6 and ADIFOR will compute this compressed Jacobian if the seed matrix is initialized to the structure shown in Figure 7.

The running time and storage requirements of the


Figure 7: Structure of ADIFOR Seed Matrix Corresponding to Compressed Jacobian in Figure 6

.

| | Sparc-2 | RS/6000 |
|---|---|---|
| Sparse DD | 0.20 | 0.130 |
| Compressed ADIFOR | 0.13 | 0.025 |
| Dense ADIFOR | 0.85 | 0.056 |

Table 1: Performance of ADIFOR-generated Derivative Code (seconds)

ADIFOR-generated code are roughly proportional to the numbers of columns of $S$, so the computation of Jacobian-vector products and compressed Jacobians requires much less time and storage than the generation of the full Jacobian matrix. For example, on the blunt body problem we observe the performance shown in Table  on Sun Microsystems SPARC 2 and IBM RS/6000-550 workstations. The first line gives the run-time of a sparse divided-difference approximation, based on the same coloring scheme as the "compressed Jacobian" approach in the second line. The third line shows the running time obtained if one treats this Jacobian as a dense one and ignores sparsity; this means that all derivative operations now are performed with vectors of length 190 instead of 28. As expected, performance suffers, although much less so on the IBM. This is due to the superscalar architecture of this chip and, we suspect, to efficient microcode implementations of multiplications by zero.

## 4 Differentiating Implicitly Defined Functions

In our experience most CFD codes in aeronautical engineering compute flow and displacements fields by

**for** $k = 1, \ldots$ **do**
  compute preconditioner $P_k$
  $z_{k+1} = z_k - P_k\, F(z_k, x_*)$
  **if** ($||F(z_k, x_*)||$ small enough) **stop**
**end for**

Figure 8: Generic Iteration for Solving $F(z_*, x_*) = 0$

iterative procedures, which may converge very slowly and often involve discontinuous adjustments of grids or free boundaries. That is, for given $x_*$ we are solving a nonlinear system

$$F(z, x_*) = 0 \tag{1}$$

to find the value $z_* = z(x_*)$ of the function implicitly defined by $F$. The question is under what circumstances an auto-differentiated version of the code implementing the rootfinding process computes the desired derivatives $z'_* = \frac{dz}{dx}|_{x=x_*}$. For the sake of discussion let us assume that our iteration for solving (1) has the form shown in Figure 8. This iteration converges when

$$||I - P_k\, F_z(z_k, x_*)|| \leq \rho < 1 \tag{2}$$

The notation $F_z$ ($F_x$) is shorthand for $\frac{\partial F}{\partial z}$ ($\frac{\partial F}{\partial x}$). Newton's method, for example, is a particular instance of this scheme with $P_k = (\frac{dF}{dz}|_{z=z_k})^{-1}$.

The implicit function theorem tells us that at the fixed-point $(z_*, x_*)$ we have

$$F_z z'_* + F_x = 0 \tag{3}$$

and in fact a not too uncommon approach (the so-called "semianalytic" approach) for obtaining $z'$ is to compute (or approximate by divided differences) $F_z(x_*)$ and $F_x(x_*)$ and to solve the resulting linear system (3) for $z'$. However, the reliability of this approach depends greatly on the conditioning of $F_z(x_*)$ as well as the accuracy of $F_z$ and $F_x$. In the following discussion we assume that a "prime" notation (like $z'$) always denotes differentiation with respect to $x$.

Applying automatic differentiation to the generic iteration of Figure 8, we obtain the iteration shown in Figure 9. Note that we have replaced the stopping criterion based on $||F||$ by one based on $\frac{dF}{dx}$. While it is natural to do so, this currently has to be done by hand. Gilbert [14] and Christianson [9] show that this iteration produces meaningful results for iterations such as Newton's method. Recently, we have been able to extend these results (details will be given in a forthcoming paper) and have shown that the simpler iteration shown in Figure 10 also converges to the desired derivative value $z'_*$ when condition (2) is satisfied. The difference between the approaches in Figures 9 and 10 is that in the latter we treat the preconditioner $P_k$ as

**for** $k = 1, \ldots$ **do**
  compute preconditioner $P_k$ and $P'_k$
  $z'_{k+1} = z'_k - P'_k F(z_k, x_*)$
  $\quad - P_k(F_z(z_k, x_*)z'_k + F_x(z_k, x_*))$
  **if** ($||F_z(z_k, x_*)z'_k + F_x(z_k, x_*)||$ small enough) **stop**
**end for**

Figure 9: Straightforward Iteration for Solving $F_z(z_*, x_*)z'_* + F_x(z_*, x_*) = 0$

**for** $k = 1, \ldots$ **do**
  compute preconditioner $P_k$
  $z'_{k+1} = z'_k - P_k\,(F_z(z_k, x_*)z'_k + F_x(z_k, x_*))$
  **if** ($||F_z(z_k, x_*)z'_k + F_x(z_k, x_*||$ small enough) **stop**
**end for**

Figure 10: Improved Iteration for Solving $F_z(z_*, x_*)z'_* + F_x(z_*, x_*) = 0$

a constant and hence drop the $P'_k F(z_k, x_*)$ term in the update of $z'_{k+1}$. This makes intuitive sense since in the end $F(z_k, x_*)$ will converge to zero anyway, thereby annihilating any contribution of $P'_k$. Also $P'_k$ is likely to involve higher derivatives that (according to the implicit function theorem) play no role in the existence of $z'_*$.

The implications of this observation for the speed of derivative computations are noteworthy. For example, in a Newton iteration, we would thus save ourselves the work of differentiating through the matrix factorization process, which is by far the dominant work of the iteration process. Exploitation of this result at the moment requires hand-modification of the ADIFOR-generated derivative code to eliminate the derivative computations for $P'_k$. Depending on code modularity, this may or may not be easy to do. We are experimenting with "deactivation" concepts which would allow this transformation to proceed in a more user-friendly fashion.

Another point worth mentioning is that it does not make sense to start the derivative iterations until the iterations for $F(z, x_*) = 0$ have essentially converged. Obviously, the derivatives $z'_*$ will not settle in until the "function value" $z_*$ itself has. Again, this requires hand-modification of the ADIFOR-generated derivative code, but the savings potential is significant. For example, Figure 11 shows the convergence behavior of the lift coefficient, which is implicitly defined as a function of the maximal airfoil thickness, the mach number, the camber, maximal camber position, and the angle of attack in Elbanna and Carlson's transonic code [13] for $M = 1.2, \alpha = 1$ with an NACA 1406 airfoil. The line

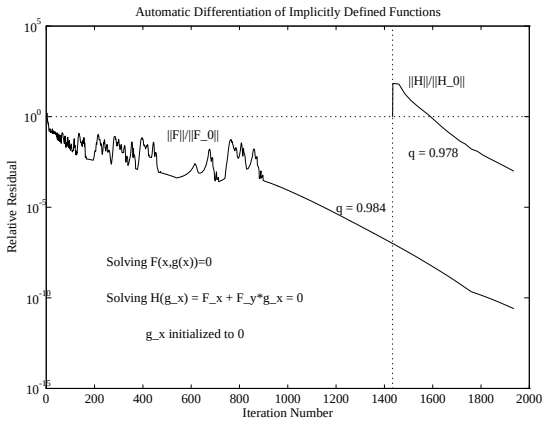


Figure 11: Convergence Behavior of Function and Derivatives in Elbanna and Carlson's Code

| Method | Time |
|---|---|
| Finite Diffs | 421 |
| Delayed ADIFOR | 217 |
| ADIFOR from the beginning | 838 |

Table 2: Run-Time of Various Derivative Schemes for Elbanna/Carlson Code on Sparc-2

labeled $||F||/||F_0||$ plots the improvement of the function residual with respect to the initial residual. We see that for the first roughly 900 iterations the nonlinear root finder hardly improves the residual. Thereafter, the convergence behavior is close to linear with a convergence factor $q \approx 0.984$. After iteration 1434, the function residual had decreased by more than 8 orders of magnitude and we started up the ADIFOR-ed version of the iteration (corresponding to the version in Figure 9), having initialized the derivatives $z_0'$ to 0. The line labelled $||H||/||H_0||$ shows the improvement in the derivative residual and we stopped after 501 more iterations when we had decreased the derivative residual by four orders of magnitude. The resulting derivative value was (up to four digits) the same as that obtained by divided difference approximations with $h = 1.0e-8$. As we can see, after the initial jump (due to our starting value), the derivatives converge linearly with a convergence rate that is comparable to that of the function iteration.

By delaying starting up the derivative iteration, we were able to realize substantial savings as shown in Table 2. The scheme just described is significantly faster than the divided-difference approximation for Elbanna and Carlson's code. On the other hand, had we not delayed computation of derivatives, and differentiated away while the function was really not converging yet, it would have taken a total of 838 seconds.

In general, the theory of automatic differentiation does so far not guarantee that the derivatives of iterates converge to the correct limits in all cases of interest. However, by monitoring the residual $||F_z(z_k, x_*)z_k' + F_x(z_k, x_*)||$, we have a constructive criterion by which to judge the progress (or stalling) of the derivative iteration process. Moreover, by judiciously deactivating certain variables one can implement various semi-analytical schemes for sensitivity analysis that have been developed and tested by aeronautical engineers (see, e.g. [22, 21, 24]. Again, derivative convergence is not a priori guaranteed, but it can be tested constructively with little extra effort.

## 5 Future Work

Our goal is to further decrease the complexity of computing derivatives, both in terms of man-hours and cpu-seconds.

For example, at the moment we are experimenting with a version of ADIFOR that in addition to computing the Jacobian values, also automatically computes the location of the nonzero entries in the Jacobian. The key observation is that all our gradient computations have the form

$$\text{vector} = \sum_i \text{scalar}_i * \text{vector}_i.$$

By merging the structure of the vectors on the right-hand side, we can obtain the structure of the vector on the left-hand side. In addition, the proper use of sparse vector data structures ensures that we perform computations only with the non-zero components of the various derivative vectors. For example, even with a relatively simple implementation of "sparse merging vectors" we obtained running times of 0.13 seconds on a Sparc-2 and 0.043 seconds on the IBM RS6000/550 for Shubin's blunt body shock problem. These numbers compare favorably with those shown in Table , in particular if one keeps in mind that *no previous knowledge of the sparsity structure was required* in the "sparse saxpy" approach. Of course the dynamic data structures required to support the automatic sparsity detection are more expensive to implement, but we found that even though the final Jacobian has at most 19 nonzeros per row, the average number of nonzeros in derivative objects was only 5.4 nonzeros. As a result, the "sparse saxpy" approach performed only 5,537 additions and 31,633 additions, whereas the "compressed Jacobian" approach performed 134,428 additions and 185,948 multiplications, and hence 88% more flops. The automatic detection of sparsity structures is a

unique capability of automatic differentiation and we expect that it will greatly contribute to the development of user-friendly optimization problem solving environments.

Another aspect that we have to address is that of input and output. ADIFOR does not yet properly handle file input and output (although there are ways to get around this [6]). The reason is that we cannot trace data dependencies through I/O statements. However, in many MDO projects the codes modeling the various disciplines have been developed and are maintained by distinct groups on different platform. They may run asynchronously and exchange data only through disk files or even magnetic tapes. For example, grid design is often done with human interaction and the result is communicated to the structural or aerodynamical code. Therefore, automatic differentiation must be modular and respect the boundaries between various system components without requiring a common runtime system. This goal has been largely achieved in ADIFOR even though automatic file transfer of derivative information is not yet possible.

In summary, we believe that as our efforts progress, automatic differentiation will be able to compute derivatives orders of magnitudes faster than current approaches and be the catalyst enabling the solution of MDO problems that were previously thought intractable.

**Acknowledgements**

We would like to thank Alan Carle, George Corliss, and Paul Hovland for many stimulating discussions and their essential roles in the ADIFOR development project. We would also like to thank Perry Newman from NASA Langley and Greg Shubin from Boeing Computing Services for making the test codes available and for sharing their knowledge with us.

**References**

[1] Christian Bischof, Alan Carle, George Corliss, and Andreas Griewank. ADIFOR: automatic differentiation in a source translator environment. ADIFOR Working Note #5, MCS–P288–0192, Mathematics and Computer Science Division, Argonne National Laboratory, 1992. Accepted for publication in Proceedings of International Symposium on Symbolic and Algebraic Computation.

[2] Christian Bischof, Alan Carle, George Corliss, Andreas Griewank, and Paul Hovland. Adifor: Generating derivative codes from Fortran programs. ADIFOR Working Note #1, MCS–P263–0991, Mathematics and Computer Science Division, Argonne National Laboratory, 1991. To appear in Scientific Programming.

[3] Christian Bischof, George Corliss, and Andreas Griewank. ADIFOR exception handling. ADIFOR Working Note #3, MCS–TM–159, Mathematics and Computer Science Division, Argonne National Laboratory, 1991.

[4] Christian Bischof, George Corliss, and Andreas Griewank. Computing second- and higher-order derivatives through univariate Taylor series. ADIFOR Working Note #6, MCS–P296–0392, Mathematics and Computer Science Division, Argonne National Laboratory, 1992.

[5] Christian Bischof and Paul Hovland. Using ADIFOR to compute dense and sparse Jacobians. ADIFOR Working Note #2, MCS–TM–158, Mathematics and Computer Science Division, Argonne National Laboratory, 1991.

[6] Christian H. Bischof, Alan Carle, George Corliss, Andreas Griewank, Paul Hovland, and Moe El-Khadiri. Getting started with adifor. ADIFOR Working Note #9, ANL–MCS–TM-164, Mathematics and Computer Science Division, Argonne National Laboratory, 1992.

[7] D. Callahan, K. Cooper, R. T. Hood, Ken Kennedy, and Linda M. Torczon. ParaScope: A parallel programming environment. *International Journal of Supercomputer Applications*, 2(4), December 1988.

[8] Alan Carle, Keith D. Cooper, Robert T. Hood, Ken Kennedy, Linda Torczon, and Scott K. Warren. A practical environment for scientific programming. *IEEE Computer*, 20(11):75–89, November 1987.

[9] Bruce Christianson. Reverse accumulation and accurate rounding error estimates for taylor series coefficients. *Optimization Methods and Software*, 1(1):81–94, 1992.

[10] Bruce D. Christianson. Automatic Hessians by reverse accumulation. Technical Report NOC TR228, The Numerical Optimisation Center, Hatfield Polytechnic, Hatfield, U.K., April 1990.

[11] T. F. Coleman, B. S. Garbow, and J. J. Moré. Software for estimating sparse Jacobian matrices. *ACM Trans. Math. Software*, 10:329 – 345, 1984.

[12] T. F. Coleman and J. J. Moré. Estimation of sparse Jacobian matrices and graph coloring problems. *SIAM Journal on Numerical Analysis*, 20:187 – 209, 1984.

[13] H.M. Elbanna and L.A. Carlson. Determination of aerodynamic sensitivity coefficients in the transonic and supersonic regimes. In *Proceedings of the*

*27th AIAA Aerospace Sciences Meeting, AIAA Paper 89-0532*. American Institute of Aeronautics and Astronautics, 1989.

[14] Jean-Charles Gilbert. Automatic differentiation and iterative processes. *Optimization Methods and Software*, 1(1):13–22, 1992.

[15] Andreas Griewank. On automatic differentiation. In M. Iri and K. Tanabe, editors, *Mathematical Programming: Recent Developments and Applications*, pages 83 – 108. Kluwer Academic Publishers, 1989.

[16] Andreas Griewank. Automatic evaluation of first- and higher-derivative vectors. In R. Seydel, F. W. Schneider, T. Küpper, and H. Troger, editors, *Proceedings of the Conference at Würzburg, Aug. 1990, Bifurcation and Chaos: Analysis, Algorithms, Applications*, volume 97, pages 135 – 148. Birkhäuser Verlag, Basel, Switzerland, 1991.

[17] Andreas Griewank and George F. Corliss, editors. *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*. SIAM, Philadelphia, Penn., 1991.

[18] Andreas Griewank and Shawn Reese. On the calculation of Jacobian matrices by the Markowitz rule. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 126 – 135. SIAM, Philadelphia, Penn., 1991.

[19] T. L. Holst, M. D. Salas, and R. W. Claus. The NASA computational aerosciences program – toward Teraflop computing. In *Proceedings of the 30th Aerospace Sciences Meeting*, pages AIAA Paper 92–0558. American Institute of Aeronautics and Astronautics, 1992.

[20] David Juedes. A taxonomy of automatic differentiation tools. In Andreas Griewank and George Corliss, editors, *Proceedings of the Workshop on Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, Philadelphia, 1991. SIAM. To appear.

[21] V. M. Korivi, A. C. Taylor, P. A. Newman, G. W. Hou, and H. E. Jones. An incremental strategy for calculating consistent discrete CFD sensitivity derivatives. NASA Technical Memorandum 104207, NASA Langley Research Center, February 1992.

[22] P. A. Newman, G. J.-W. Hou, H. E. Jones, A. C. Taylor, and V. M. Korivi. Observations on computational methodologies for use in large-scale, gradient-based, multidisciplinary design incorporating advanced CFD codes. NASA Technical Memorandum 104206, NASA Langley Research Center, 1992.

[23] Louis B. Rall. *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1981.

[24] G. R. Shubin. Obtaining "cheap" optimization gradients from computational aerodynamics codes. Applied Mathematics and Statistics Technical Report AMS–TR–164, Boeing Computer Services, June 1991.

[25] G. R. Shubin, A. B. Stephens, H. M. Glaz, A. B. Wardlaw, and L. B. Hackerman. Steady shock tracking, Newton's method, and the supersonic blunt body problem. *SIAM J. on Sci. and Stat. Computing*, 3(2):127 – 144, June 1982.